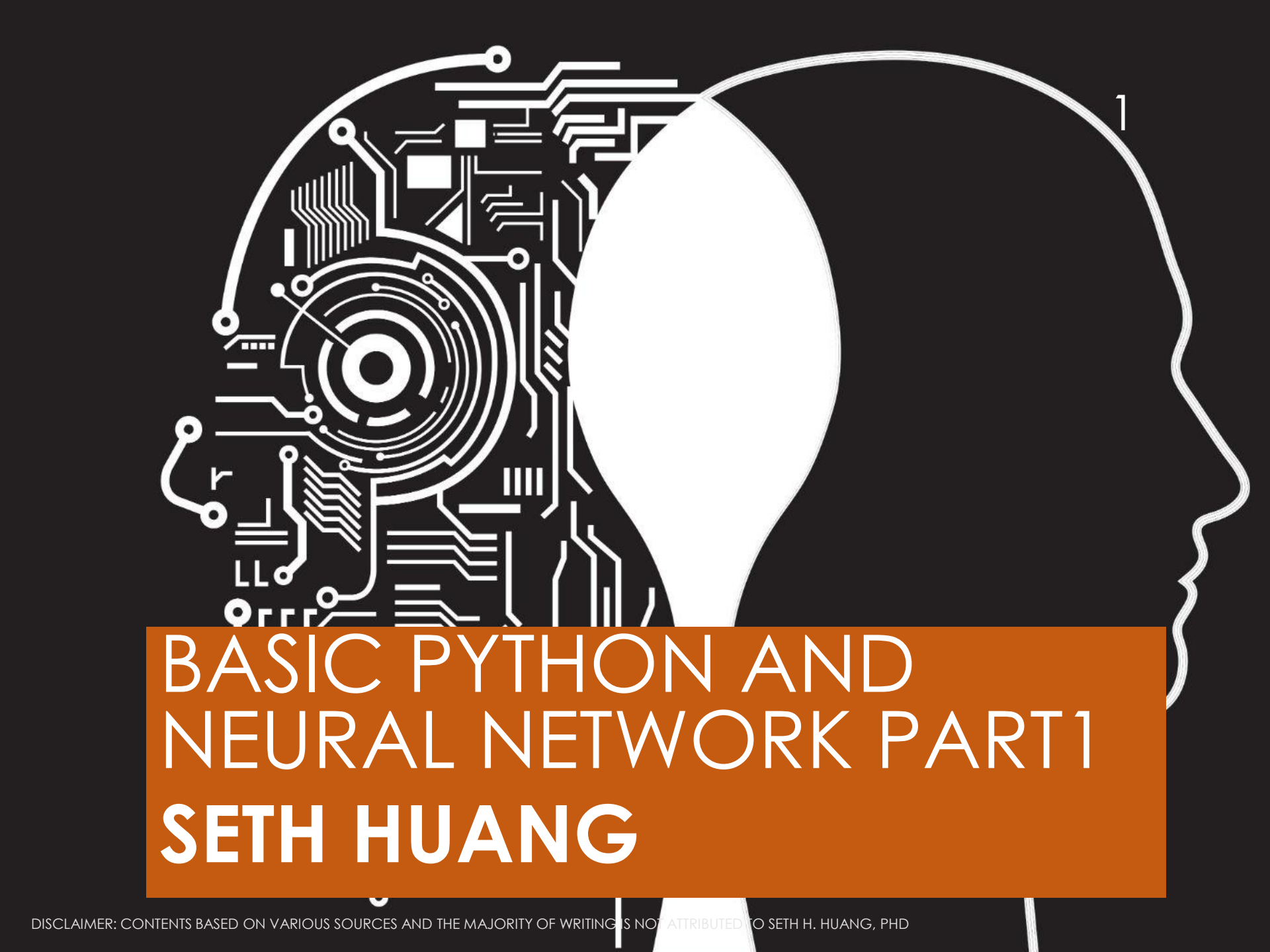


BASIC PYTHON AND NEURAL NETWORK PART1

SETH HUANG

- Installing Python through Anaconda
- Install Sublime
- Neural Network Structure
- Basic programming intro

BASIC NEURAL NETS

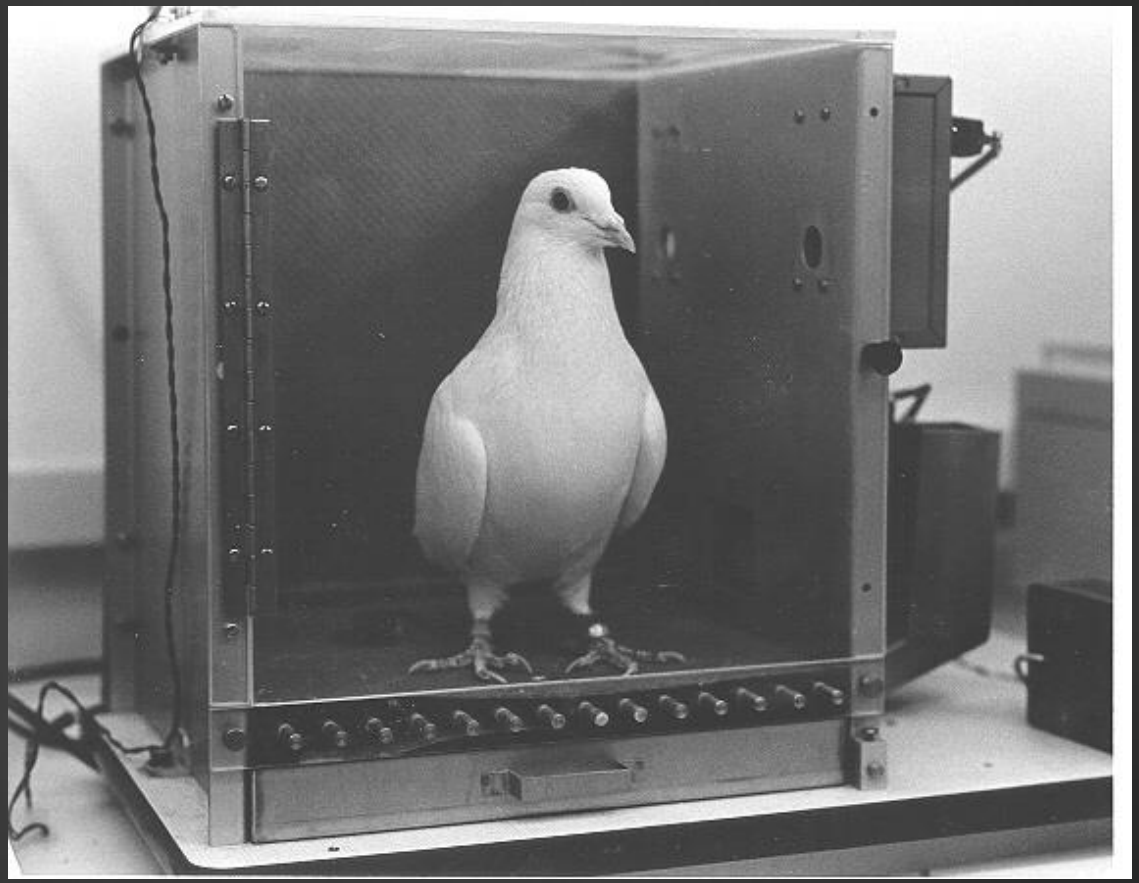
What are Neural Networks?

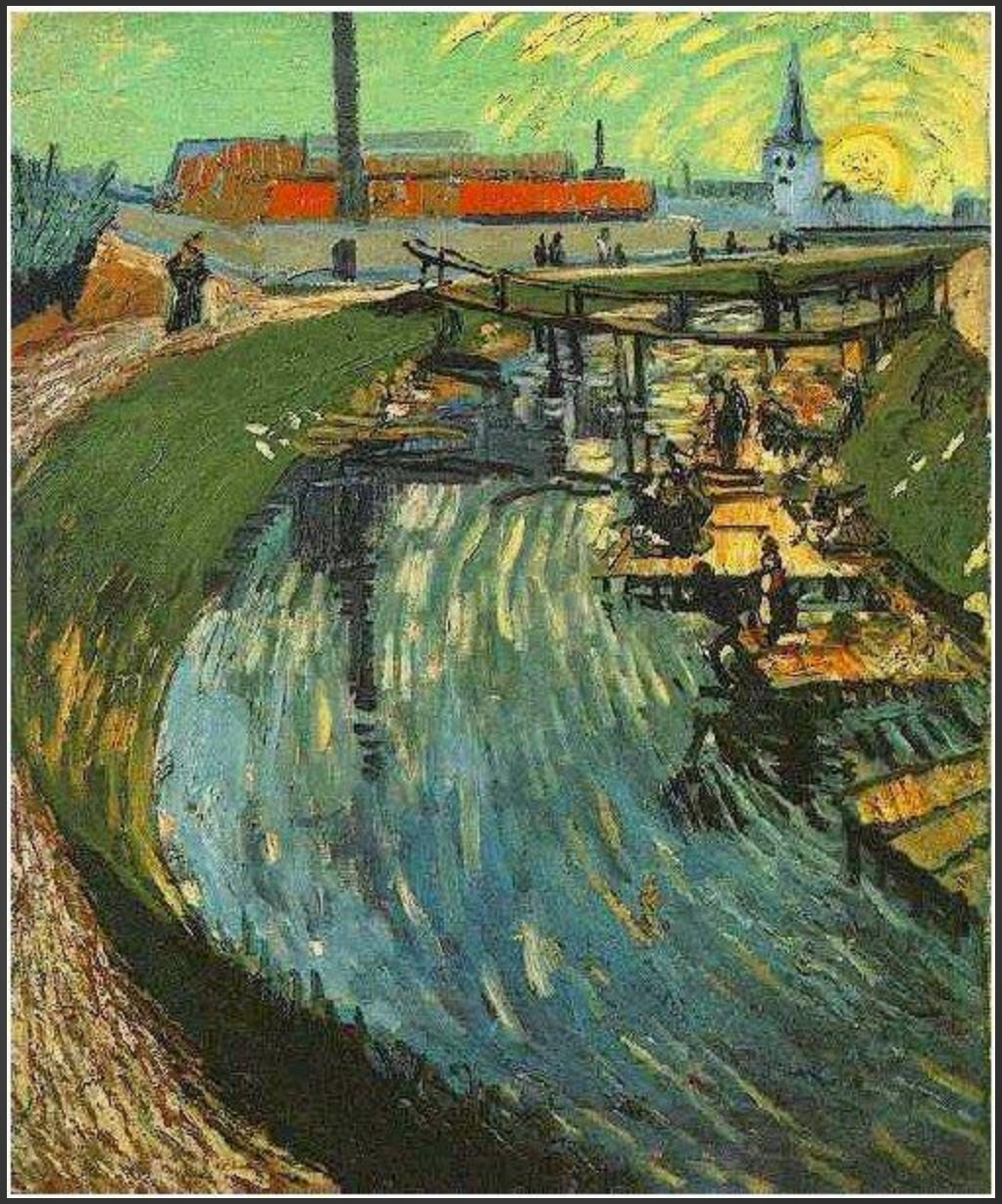
- Models of the brain and nervous system
- Highly parallel
  - Process information much more like the brain than a serial computer (different features at the same time)
- Learning biology
- Very simple principles
- But very complex behaviours
- Applications
  - As powerful problem solvers
  - As biological models

Biological Neural Nets: Inspiration

- Pigeons as art experts (Watanabe *et al.* 1995)
  - Experiment:
    - Pigeon in Skinner box
    - Present paintings of two different artists (e.g. Chagall / Van Gogh)
    - Reward for pecking when presented a particular artist (e.g. Van Gogh)

Are Pigeons Smarter than You?





The Results

- Pigeons were able to discriminate between Van Gogh and Chagall with 95% accuracy (when presented with pictures they had been trained on)

- Discrimination still 85% successful for previously unseen paintings of the artists

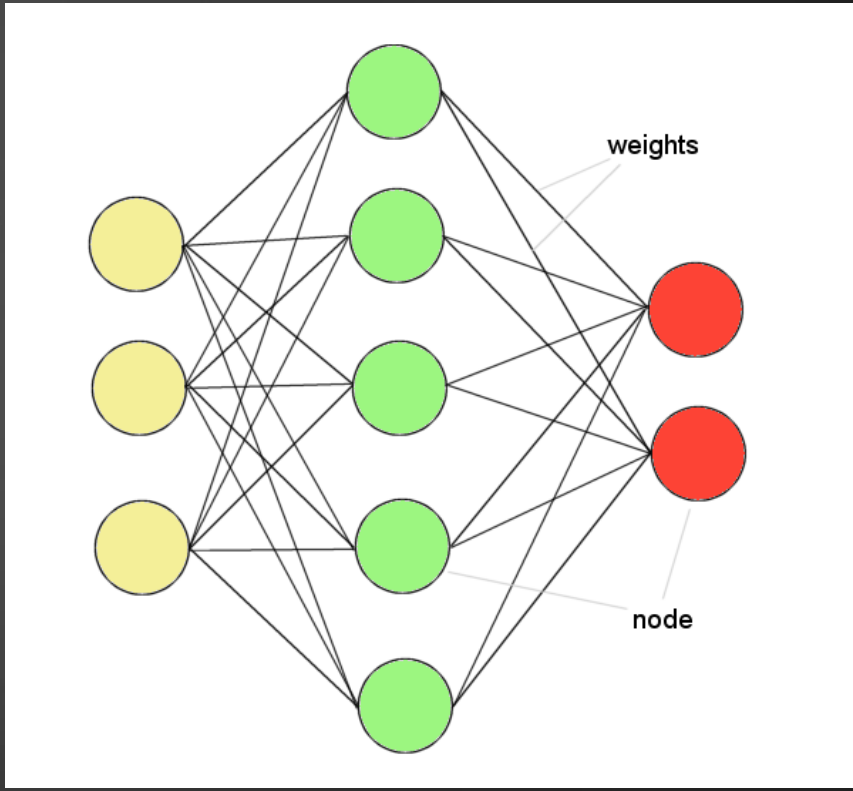
- Pigeons do not simply memorize the pictures
- They can extract and recognize patterns (the 'style')
- They generalize from the already seen to make predictions

- This is what neural networks (biological and artificial) are good at (unlike conventional computer)

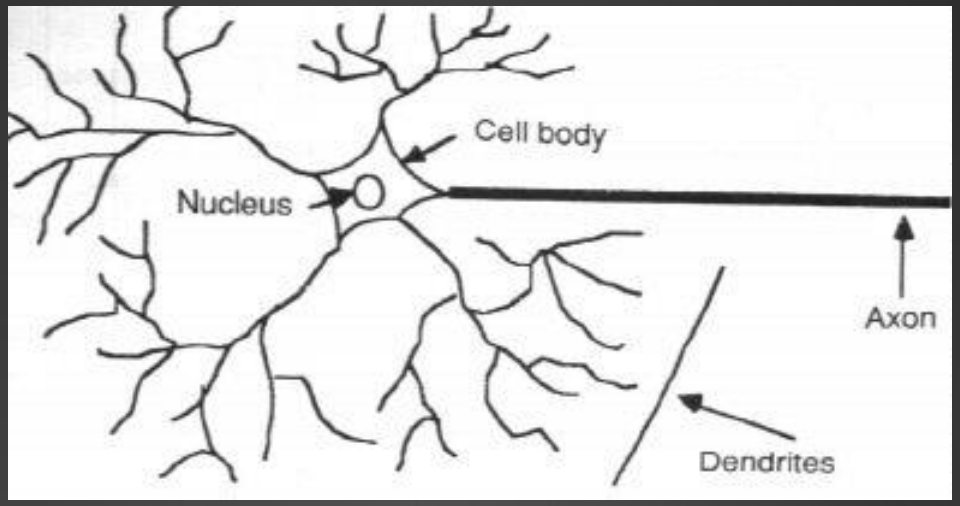
ANNs – The basics

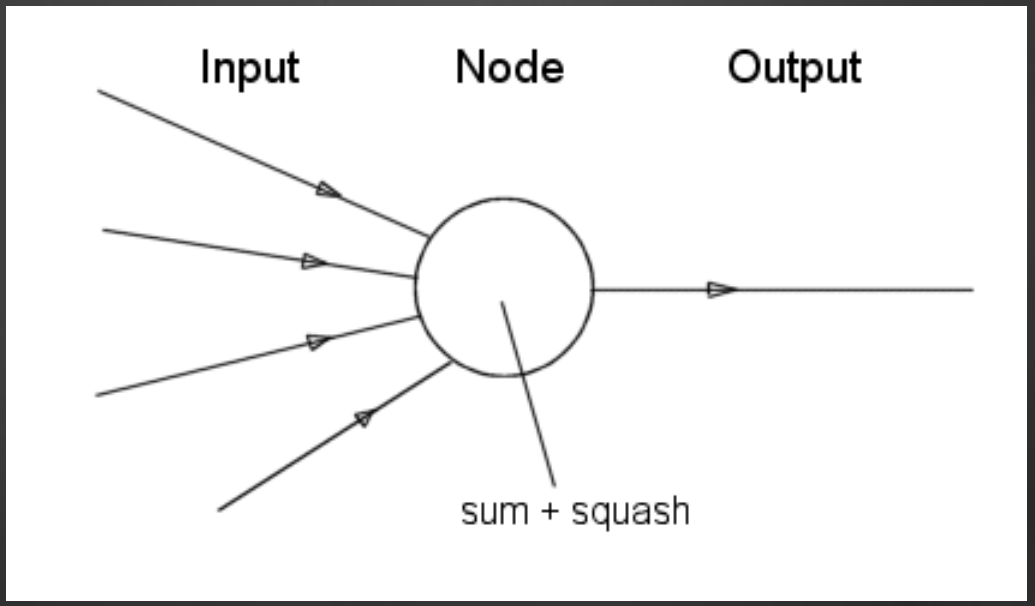
- ANNs incorporate the two fundamental components of biological neural nets:

1. Neurones (nodes)
2. Synapses (weights)

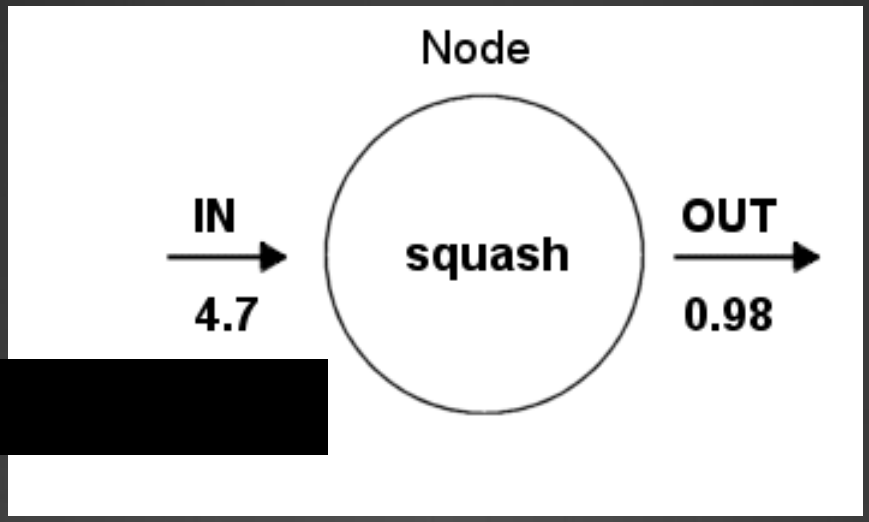


- Neurone vs. Node

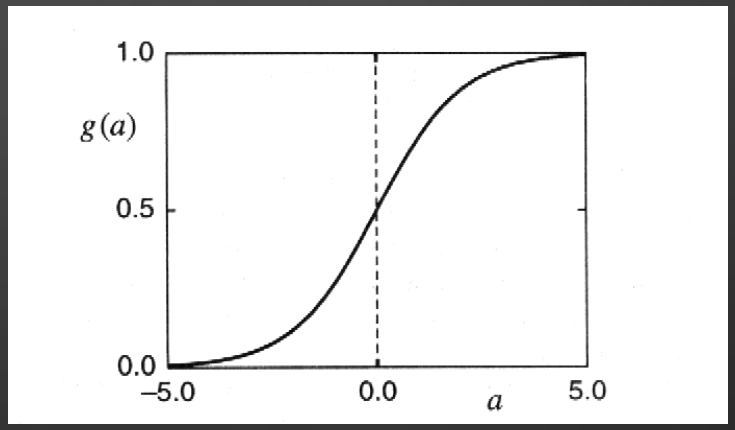




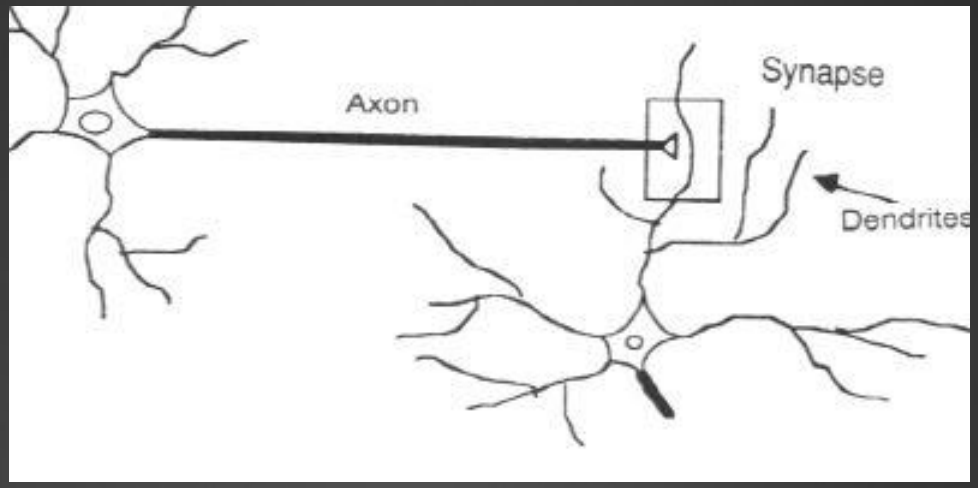
- Structure of a node:

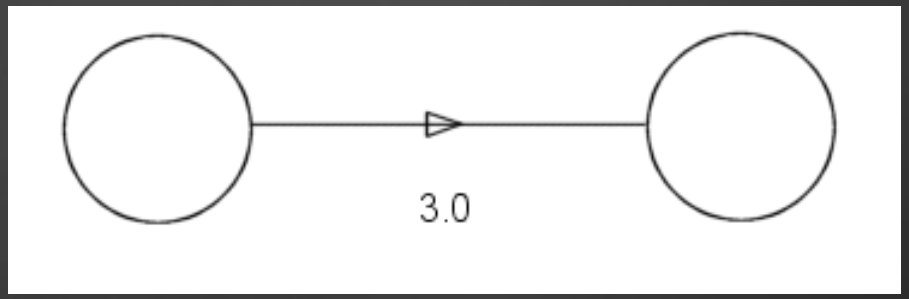


Squashing function limits node output (activation function):



Synapse vs. weight





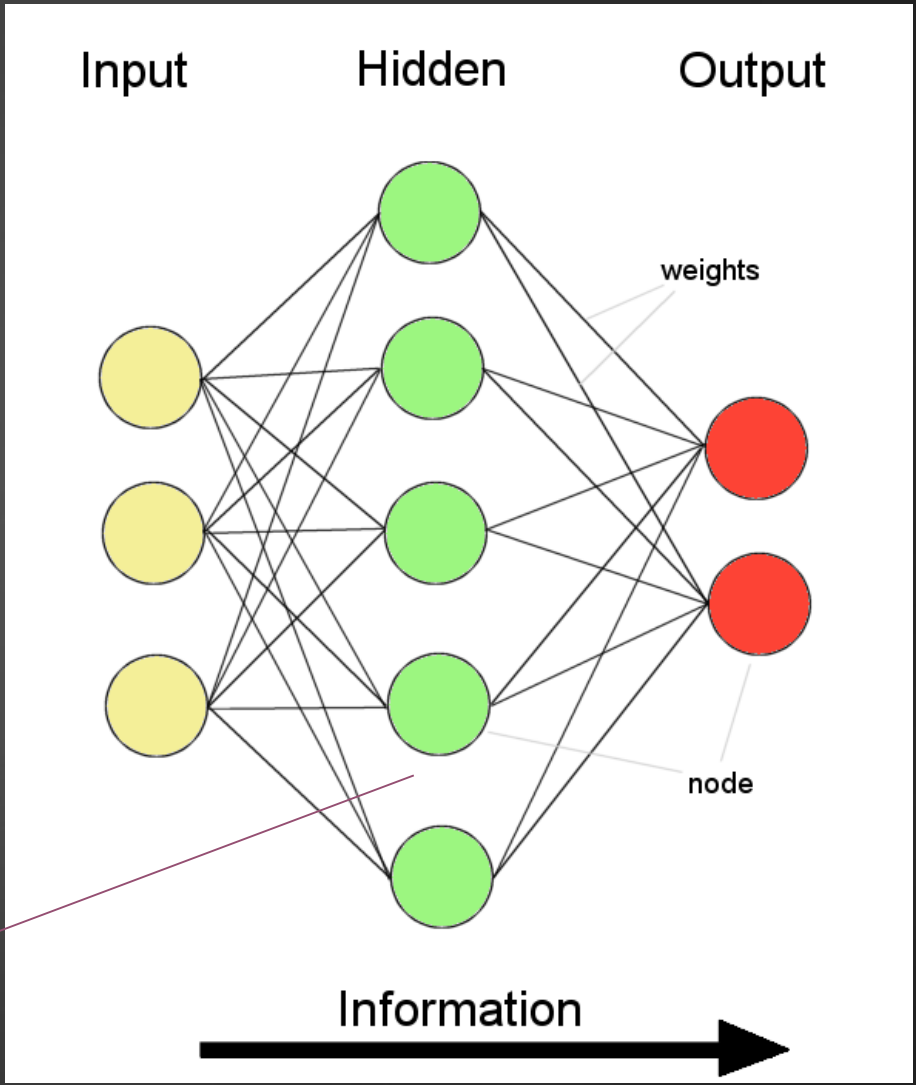
Feed-forward nets

Information flow is unidirectional

- Data is presented to *Input layer*
- Passed on to *Hidden Layer*
- Passed on to *Output layer*

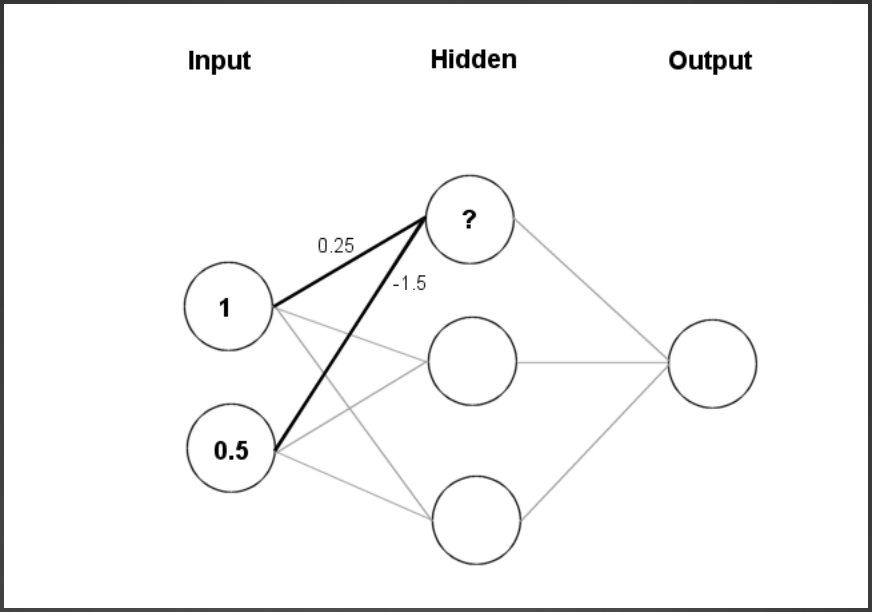
Information is distributed

Information processing is parallel



Internal representation (interpretation) of data

- Feeding data through the net:



$(1 \times 0.25) + (0.5 \times (-1.5)) = 0.25 + (-0.75) = -\mathbf{0.5}$

Squashing: $\frac{1}{1+e^{0.5}} = 0.3775$

- Data is presented to the network in the form of activations in the input layer

- Examples
  - Pixel intensity (for pictures)
  - Molecule concentrations (for artificial nose)
  - Share prices (for stock market prediction)

- Data usually requires preprocessing
  - Analogous to senses in biology

- How to represent more abstract data, e.g. a name?
  - Choose a pattern, e.g.
    - 0-0-1 for “Chris”
    - 0-1-0 for “Becky”

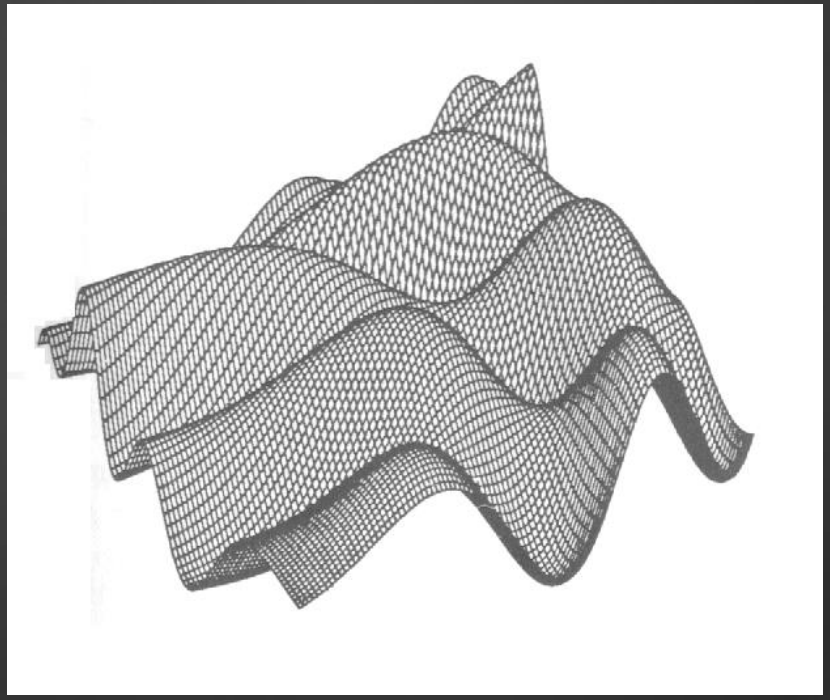
- Weight settings determine the behaviour of a network

→ How can we find the right weights?

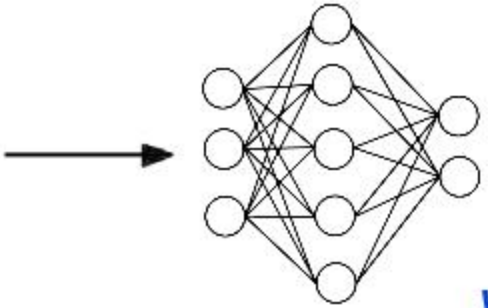
Training the Network - Learning

- Backpropagation
  - Requires training set (input / output pairs)
  - Starts with small random weights
  - Error is used to adjust weights (supervised learning)

  → Gradient descent on error landscape



1.

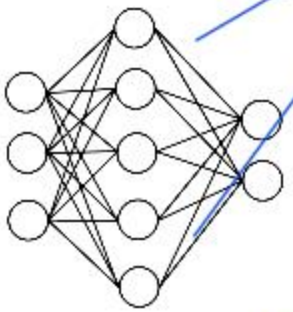


Wallace

Wallace - Darwin (calculate error)

adjust weights

2.

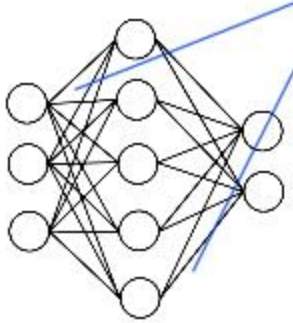


Wallace

Wallace - Darwin (calculate error)

adjust weights

3.



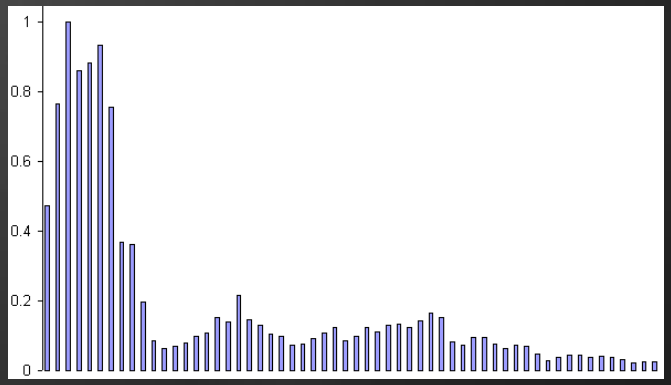
Darwin

- **Advantages**
  - It works!
  - Relatively fast
- **Downsides**
  - Requires a training set
  - Can be slow as you increase the number of layers
  - Probably not biologically realistic
- **Alternatives to Backpropagation**
  - Reinforcement learning
    - Only limited success UNTIL RECENTLY
  - Artificial evolution (genetic algo)
    - More general, but can be even slower than backprop

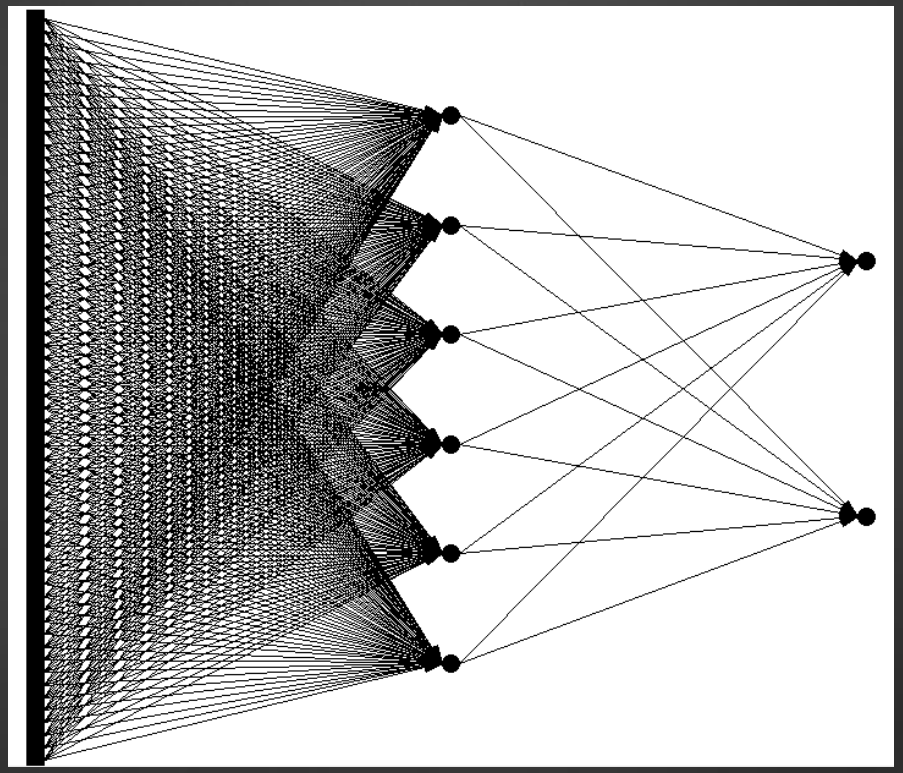
Example: Voice Recognition

- Task: Learn to discriminate between two different voices saying "Hello"

- Data
  - Sources (two researchers)
    - Steve Simpson
    - David Raubenheimer
  - Format
    - Frequency distribution (60 bins)
    - Analogy: cochlea

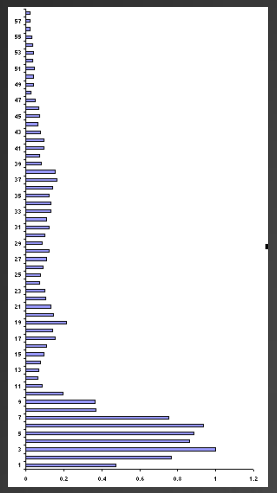


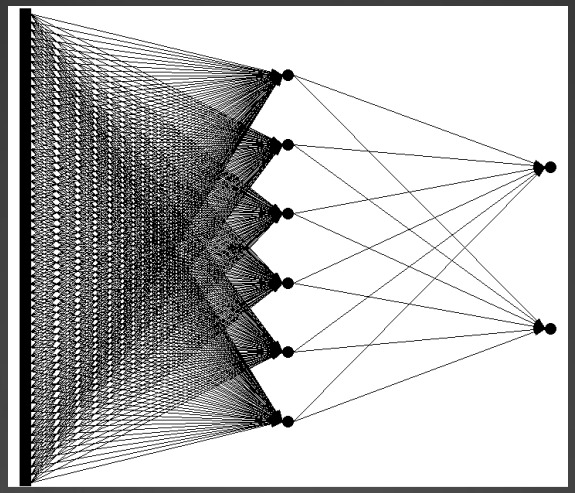
- Network architecture
  - Feed forward network
    - 60 input (one for each frequency bin)
    - 6 hidden
    - 2 output (0-1 for “Steve”, 1-0 for “David”)



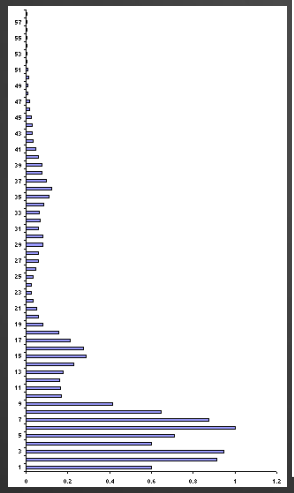
- Presenting the data

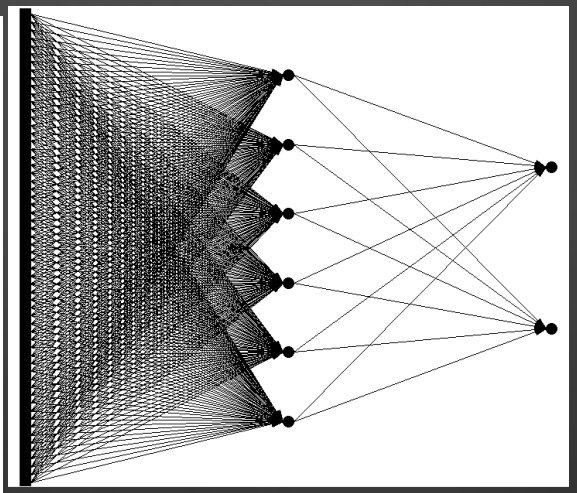
Steve





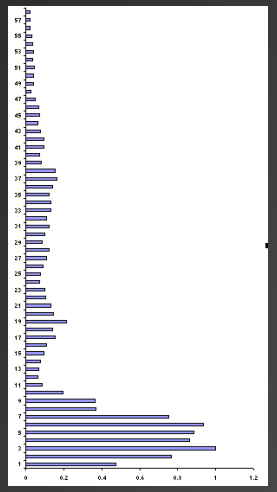
David

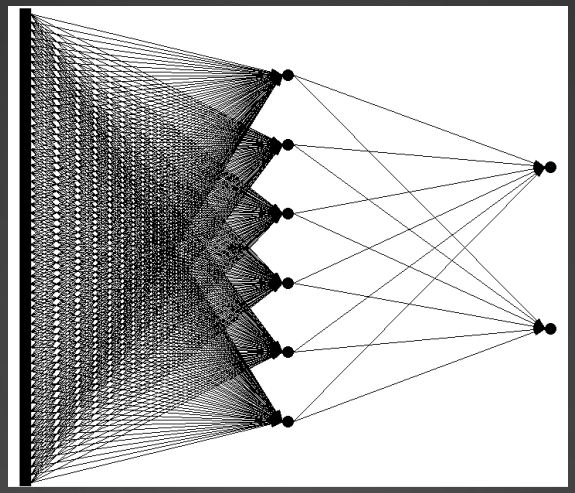




- Presenting the data (untrained network)

Steve

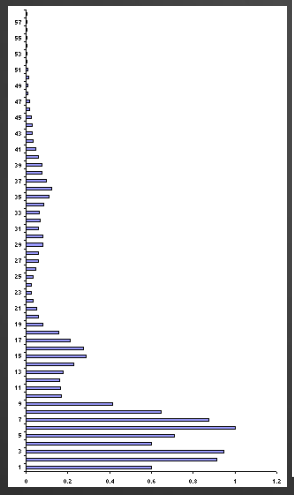


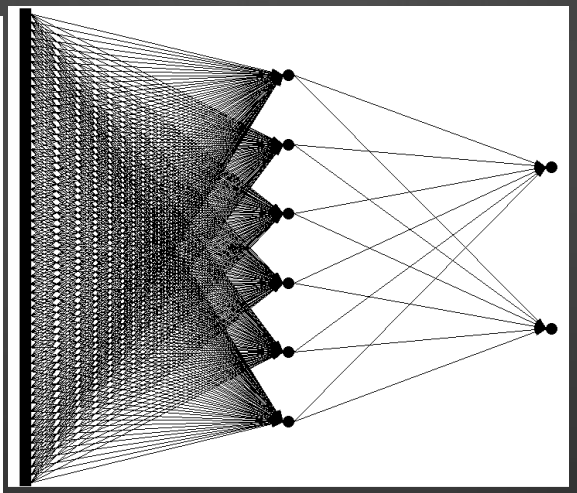


0.43

0.26

David



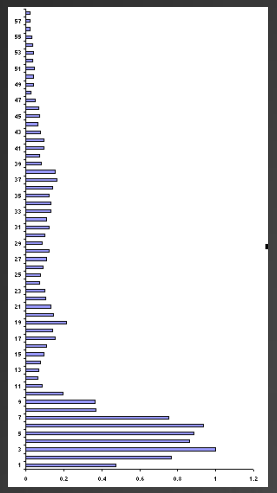


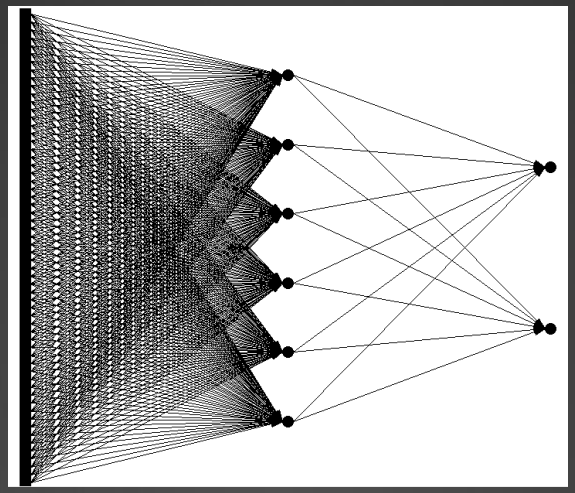
0.73

0.55

- Calculate error

Steve

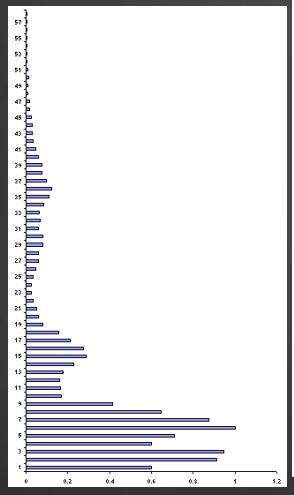


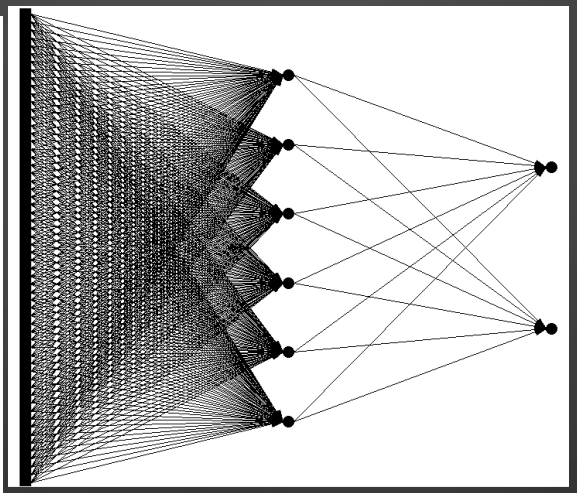


$0.43 - 0 = 0.43$

$0.26 - 1 = 0.74$

David



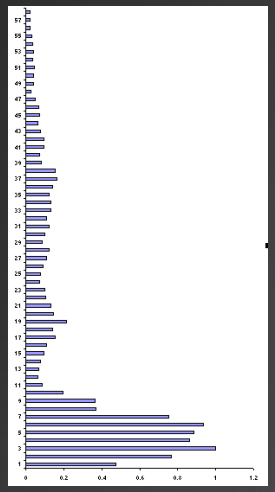


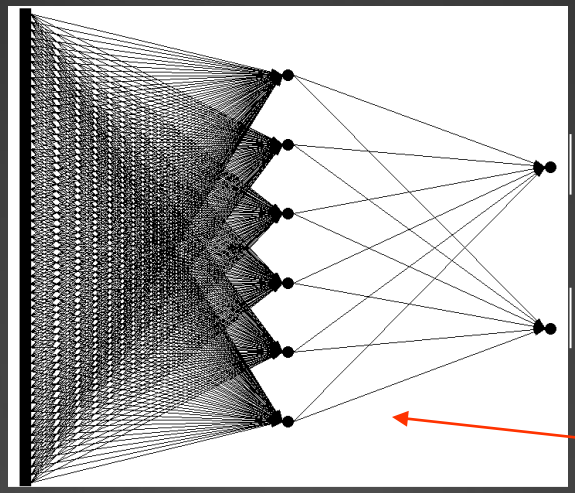
$0.73 - 1 = 0.27$

$0.55 - 0 = 0.55$

- Backprop error and adjust weights

Steve



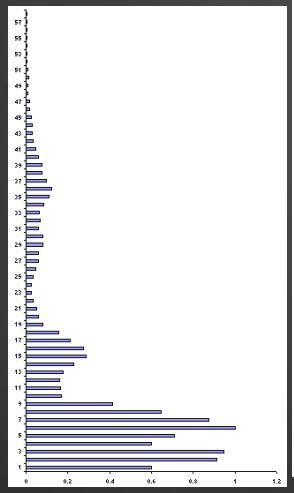


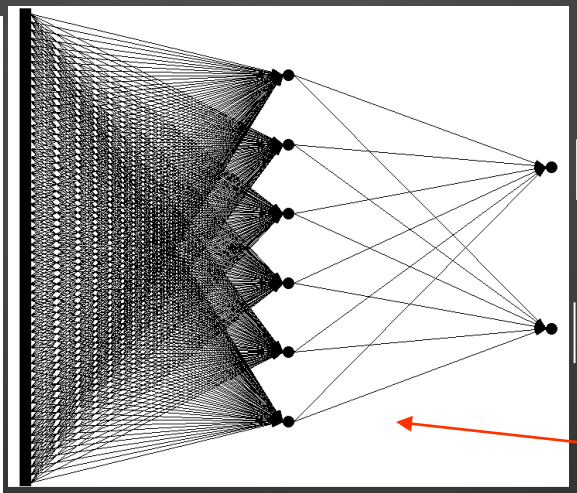
$$|0.43 - 0| = 0.43$$

$$|0.26 - 1| = 0.74$$

$$1.17$$

David





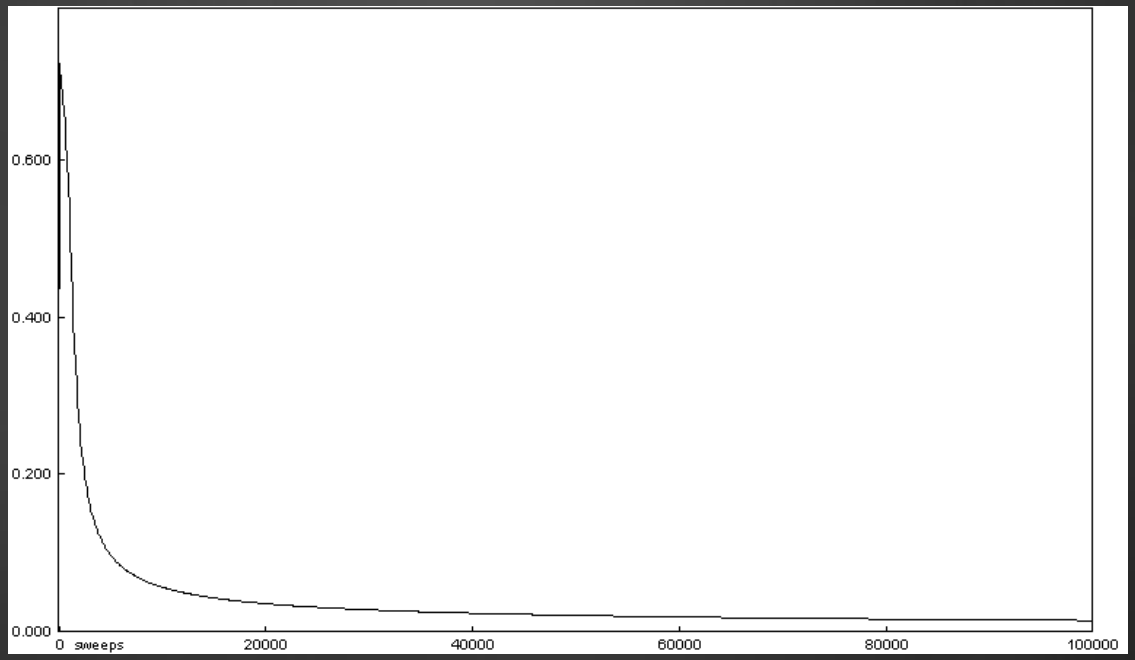
$$|0.73 - 1| = 0.27$$

$$|0.55 - 0| = 0.55$$

$$0.82$$

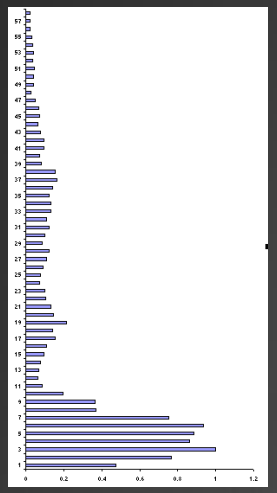
- Repeat process (sweep) for all training pairs
  - Present data
  - Calculate error
  - Backpropagate error
  - Adjust weights

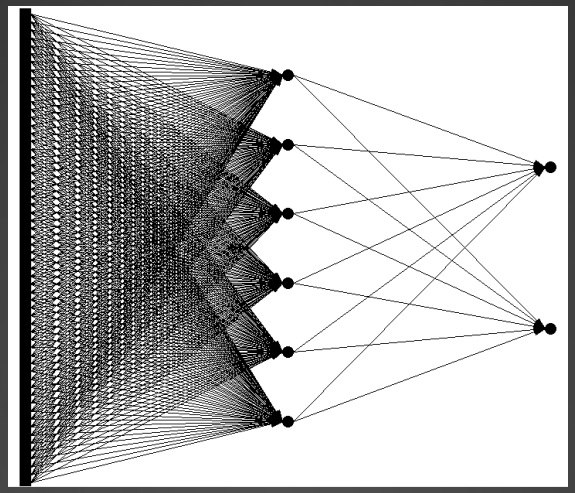
- Repeat process multiple times



- Presenting the data (trained network)

Steve

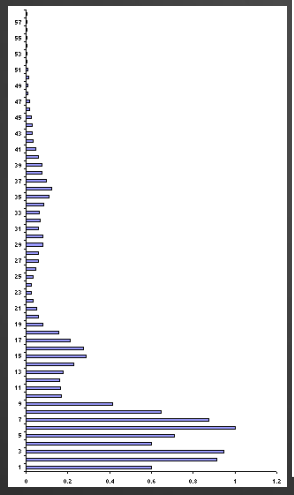


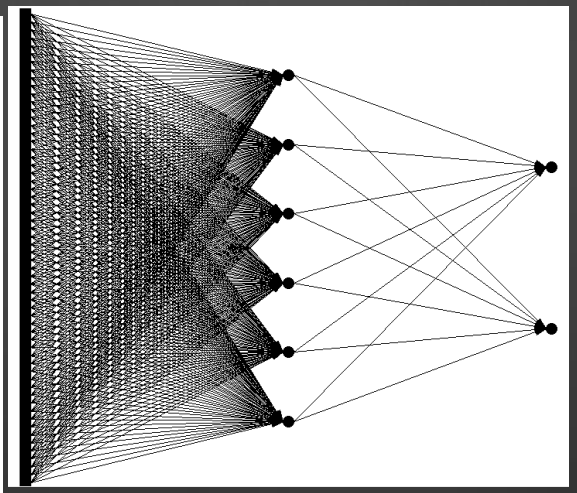


0.01

0.99

David





0.99

0.01

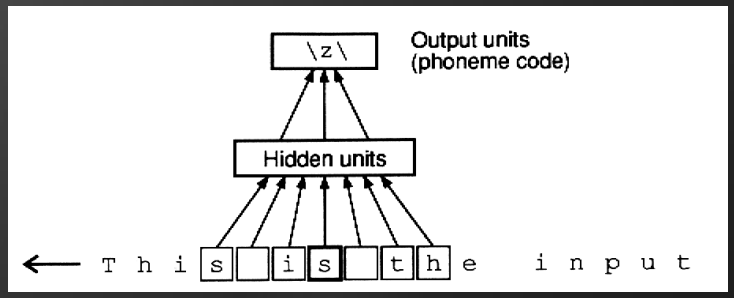
- Results – Voice Recognition
  - Performance of trained network
    - Discrimination accuracy between known "Hello"s
      - 100%
    - Discrimination accuracy between new "Hello"'s
      - 100%

- Results – Voice Recognition (ctnd.)
  - Network has learnt to generalise from original data
  - Networks with different weight settings can have same functionality
  - Trained networks ‘concentrate’ on lower frequencies
  - Network is robust against non-functioning nodes

Applications of Feed-forward nets

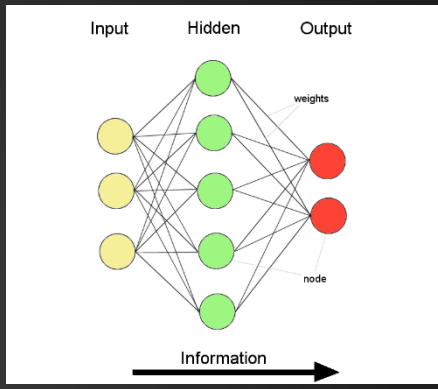
- Pattern recognition
  - Character recognition
  - Face Recognition
- Sonar mine/rock recognition (Gorman & Sejnowksi, 1988)
- Navigation of a car (Pomerleau, 1989)
- Stock-market prediction
- Pronunciation (NETtalk)

  Yes, a bit creepy.



Recurrent Networks

- Feed forward networks:
  - Information only flows one way
  - One input pattern produces one output
  - No sense of time (or memory of previous state)



- Recurrency
  - Nodes connect back to other nodes or themselves
  - Information flow is multidirectional
  - Sense of time and memory of previous state(s)

- Biological nervous systems show high levels of recurrency (but feed-forward structures exists too)

Classic experiment on language acquisition and processing (Elman, 1990)

- **Task**
  - Elman net to predict successive words in sentences.
- **Data**
  - Suite of sentences, e.g.
    - "The boy catches the ball."
    - "The girl eats an apple."
  - Words are input one at a time
- **Representation**
  - Binary representation for each word, e.g.
    - 0-1-0-0-0 for "girl"
- **Training method**
  - Backpropagation

In Conclusion, from now until June, we will be dealing with:

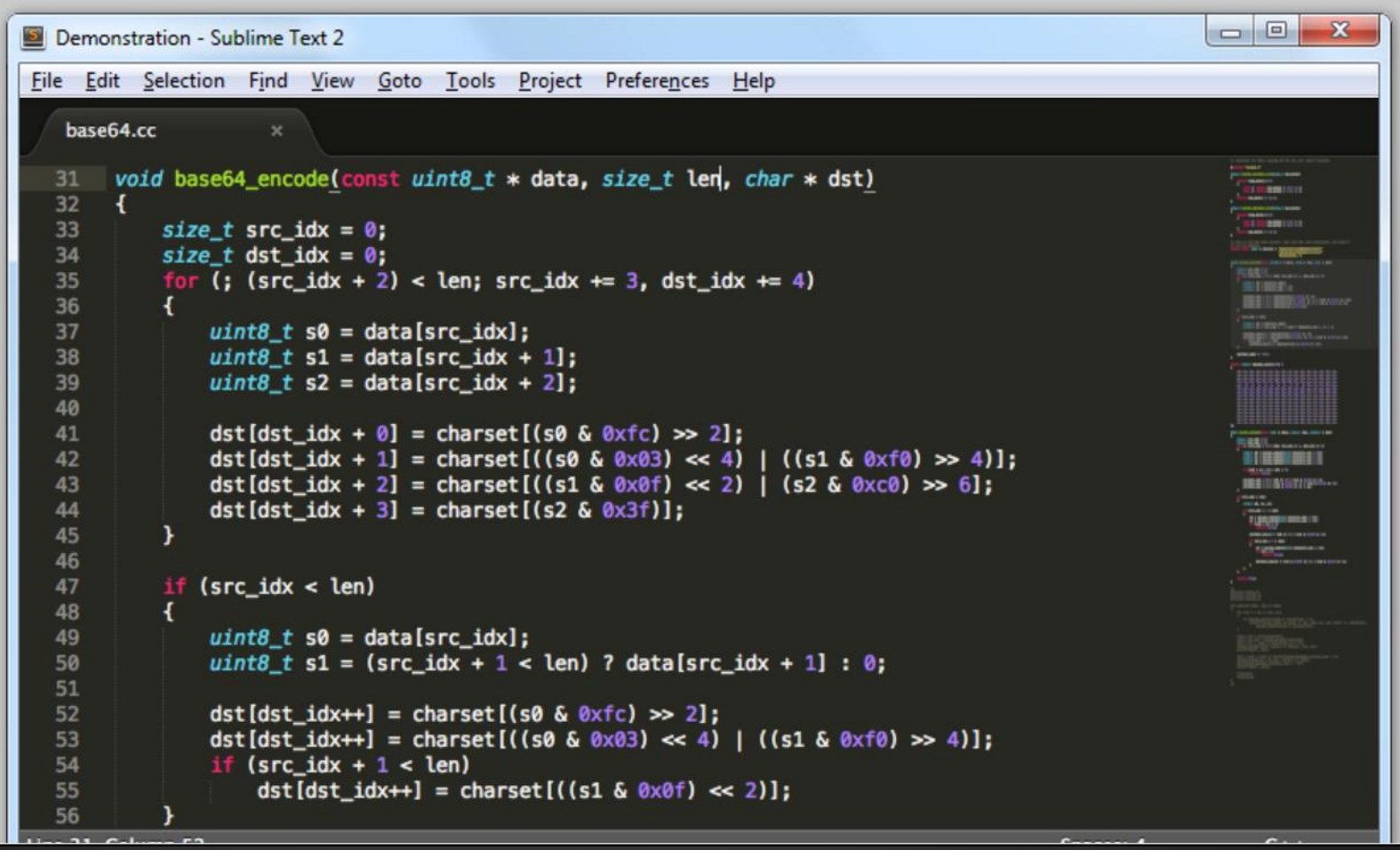
1. Neural Nets, layers, nodes, weights => Deep Learning
2. Backpropagation
3. Stochastic Gradient Descent
4. Lastly, Convolution Neural Network

1. Neural Network with Python
⇒ Recognizing MNIST digits
⇒ Packages: numpy
2. Basic Neural Construction:
⇒ Packages: Scilearn
3. Deep Learning/ Convolution Neural Network (Hopefully)
⇒Packages: Theano

Before that, we have to know the very basic things about programming, so we will have a crash course on Python (this week and next two weeks)

Sublime Text

Sublime Text is a sophisticated text editor for code, markup and prose.
You'll love the slick user interface, extraordinary features and amazing performance.



Why Sublime?

- Python is an interpretative language.

- You can write python codes in text, and let the interpreter run it.
=> Many different programs and methods.

We are using the most popular packages and most popular programs:
- ipython notebook (Anaconda)
- Sublime (or Note ++)
- Pycharm (IDE)
- Maybe Terminal.com (Cloud computing)

Which of these languages do you know?

- C or C++
- Java
- Perl
- Scheme
- Fortran
- Python
- Matlab

- Running Python and Output
- Data Types
- Input and File I/O
- Control Flow
- Functions
- Then, Why Python in Scientific Computation?
- Binary distributions Scientific Python

Hello World

- Open a terminal window and type “python”
- If on Windows open a Python IDE like IDLE
- At the prompt type ‘hello world!’

TRY IT ON YOUR COMPUTER

```
>>> 'hello world!'
'hello world!'
```

Python Overview

From *Learning Python, 2nd Edition*:

- Programs are composed of modules
- Modules contain statements
- Statements contain expressions
- Expressions create and process objects

What does that mean?

The Python Interpreter

- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print 'print me'
print me
>>>
```

The print Statement

- Elements separated by commas print with a space between them
- A comma at the end of the statement (print ‘hello’,) will not print a newline character

```
>>> print 'hello'
hello
>>> print 'hello', 'there'
hello there
```

Documentation

The ‘#’ starts a line comment

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

Variables

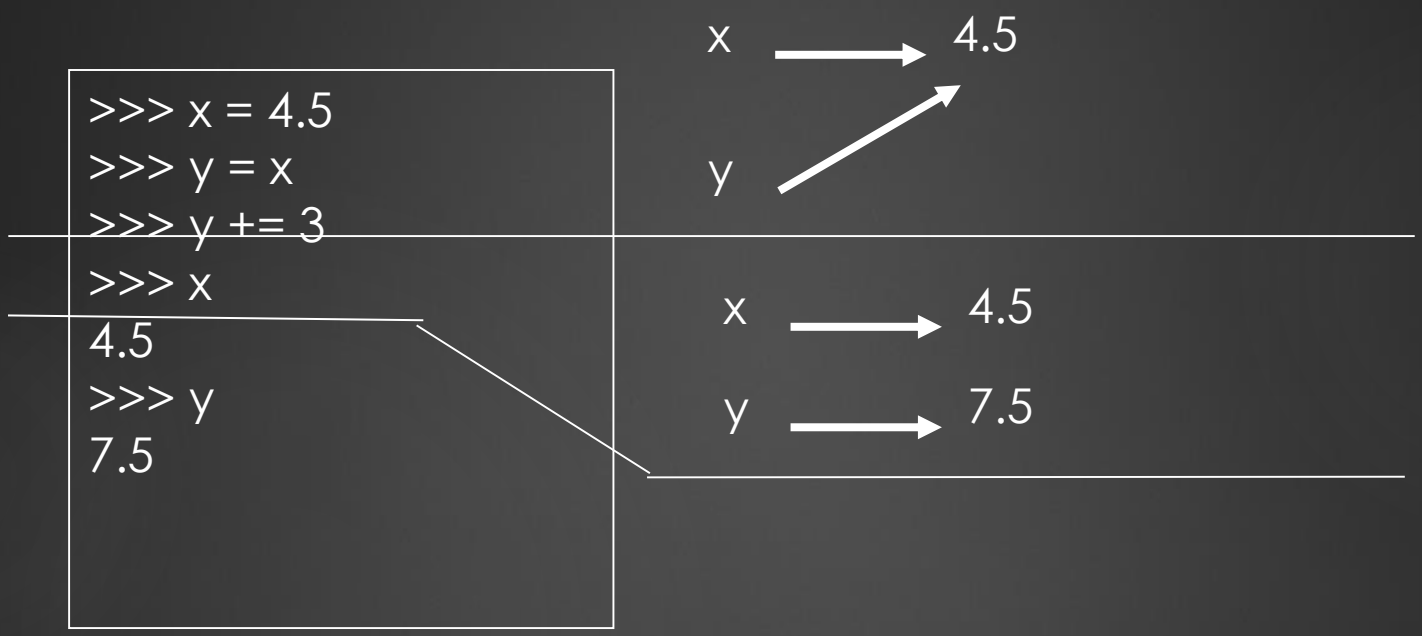
- Are not declared, just assigned
- The variable is created the first time you assign it a value
- Are references to objects
- Type information is with the object, not the reference
- Everything in Python is an object

Numbers: Floating Point

- int(x) converts x to an integer
- float(x) converts x to a floating point
- The interpreter shows a lot of digits

```
>>> 1.23232
1.2323200000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

Numbers are *immutable*



String Literals: Many Kinds

- Can use single or double quotes, and three double quotes for a multi-line string

```
>>> 'I am a string'
'I am a string'
>>> "So am I!"
'So am I!'
>>> s = """And me too!
though I am much longer
than the others :)"""
'And me too!\nthough I am much longer\nthan the
others :)‘
>>> print s
And me too!
though I am much longer
than the others :)‘
```

Substrings and Methods

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- **len**(String) – returns the number of characters in the String
- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>> str(10.3)
'10.3'
```

String Formatting

- Similar to C's printf
- <formatted string> % <elements to insert>
- Can usually just use %s for everything, it will convert the object to its String representation.

```
>>> "One, %d, three" % 2
'One, 2, three'
>>> "%d, two, %s" % (1,3)
'1, two, 3'
>>> "%s two %s" % (1, 'three')
'1 two three'
>>>
```

Lists

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Lists: Modifying Content

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Lists: Modifying Contents

- The method **append** modifies the list and returns **None**
- List addition (**+**) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```

Tuples

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:

  ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Dictionaries

- A set of key-value pairs
- Dictionaries are *mutable*

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Dictionaries: Add/Modify

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah': [1, 2, 3]}
```

Dictionaries: Deleting Elements

- The **del** method deletes an element from a dictionary

```
>>> d
{1: 'hello', 2: 'there', 10: 'world'}
>>> del(d[2])
>>> d
{1: 'hello', 10: 'world'}
```

Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Data Type Summary

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable
- All variables are references

Data Type Summary

- Integers: 2323, 3234L
- Floating Point: 32.3, 3.1E2
- Complex: 3 + 2j, 1j
- Lists: l = [1,2,3]
- Tuples: t = (1,2,3)
- Dictionaries: d = {‘hello’ : ‘there’, 2 : 15}

Input

- The **raw_input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Input: Example

```
print "What's your name?"
name = raw_input("> ")

print "What year were you born?"
birthyear = int(raw_input("> "))

print "Hi %s! You are %d years old!" % (name, 2011 - birthyear)
```

```
~: python input.py
What's your name?
> Michael
What year were you born?
>1980
Hi Michael! You are 31 years old!
```

Moving to Files

- The interpreter is a good place to try out some code, but what you type is not reusable
- Python code files can be read into the interpreter using the **import** statement

No Braces

- Python uses ***indentation*** instead of braces to determine the scope of expressions
- All lines must be indented the same amount to be part of the scope (or indented more if part of an inner scope)
- This **forces** the programmer to use proper indentation since the indenting is part of the program!

If Statements

```
import math
x = 30
if x <= 15 :
    y = x + 15
elif x <= 30  :
    y = x + 30
else :
    y = x
print 'y = ',
print math.sin(y)
```

In file ifstatement.py

```
>>> import ifstatement
y =  0.999911860107
>>>
```

In interpreter

While Loops

```
x = 1
while x < 10 :
    print x
    x = x + 1
```

In whileloop.py

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

Loop Control Statements

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

The Loop Else Clause

- The optional **else** clause runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print x
    x = x + 1
else:
    print 'hello'
```

In whileelse.py

```
~: python whileelse.py
1
2
hello
```

Run from the command line

The Loop Else Clause

```
x = 1
while x < 5 :
    print x
    x = x + 1
    break
else :
    print 'i got here'
```

whileelse2.py

```
~: python whileelse2.py
1
```

For Loops

- **For** loops also may have the optional **else** clause

```
for x in range(5):
    print x
    break
else :
    print 'i got here'
```

elseforloop.py

```
~: python elseforloop.py
1
```

Function Basics

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

functionbasics.py

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

Functions are first class objects

- Can be assigned to a variable
- Can be passed as a parameter
- Can be returned from a function
- Functions are treated like any other variable in Python, the **def** statement simply assigns a function to a variable

Function names are like any variable

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Parameters: Defaults

- Parameters can be assigned default values
- They are overridden if a parameter is given for them
- The type of the default doesn't limit the type of a parameter

```
>>> def foo(x = 3) :
...     print x
...
>>> foo()
3
>>> foo(10)
10
>>> foo('hello')
hello
```

Parameters: Named

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...     print a, b, c
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

Modules

- The highest level structure of Python
- Each file with the py suffix is a module
- Each module has its own namespace

Exercise: Comments and Characters


```
1  # A comment, this is so you can read your program later.
2  # Anything after the # is ignored by python.
3
4  print "I could have code like this." # and the comment after is ignored
5
6  # You can also use a comment to "disable" or comment out a piece of code:
7  # print "This won't run."
8
9  print "This will run."
```

```
$ python ex2.py
I could have code like this.
This will run.
```

Exercise: Nums and Math


```
1  print "I will now count my chickens:"
2
3  print "Hens", 25 + 30 / 6
4  print "Roosters", 100 - 25 * 3 % 4
5
6  print "Now I will count the eggs:"
7
8  print 3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6
9
10 print "Is it true that 3 + 2 < 5 - 7?"
11
12 print 3 + 2 < 5 - 7
13
14 print "What is 3 + 2?", 3 + 2
15 print "What is 5 - 7?", 5 - 7
16
17 print "Oh, that's why it's False."
18
19 print "How about some more."
20
21 print "Is it greater?", 5 > -2
22 print "Is it greater or equal?", 5 >= -2
23 print "Is it less or equal?", 5 <= -2
```

Exercise: Variables and Names


```
1  cars = 100
2  space_in_a_car = 4.0
3  drivers = 30
4  passengers = 90
5  cars_not_driven = cars - drivers
6  cars_driven = drivers
7  carpool_capacity = cars_driven * space_in_a_car
8  average_passengers_per_car = passengers / cars_driven
9
10
11 print "There are", cars, "cars available."
12 print "There are only", drivers, "drivers available."
13 print "There will be", cars_not_driven, "empty cars today."
14 print "We can transport", carpool_capacity, "people today."
15 print "We have", passengers, "to carpool today."
16 print "We need to put about", average_passengers_per_car, "in each car."
```

Exercise: More Variables and Printing


```
1  my_name = 'Zed A. Shaw'
2  my_age = 35 # not a lie
3  my_height = 74 # inches
4  my_weight = 180 # lbs
5  my_eyes = 'Blue'
6  my_teeth = 'White'
7  my_hair = 'Brown'
8
9  print "Let's talk about %s." % my_name
10 print "He's %d inches tall." % my_height
11 print "He's %d pounds heavy." % my_weight
12 print "Actually that's not too heavy."
13 print "He's got %s eyes and %s hair." % (my_eyes, my_hair)
14 print "His teeth are usually %s depending on the coffee." % my_teeth
15
16 # this line is tricky, try to get it exactly right
17 print "If I add %d, %d, and %d I get %d." % (
18     my_age, my_height, my_weight, my_age + my_height + my_weight)
```

Exercise: Strings and Text


```
1  x = "There are %d types of people." % 10
2  binary = "binary"
3  do_not = "don't"
4  y = "Those who know %s and those who %s." % (binary, do_not)
5
6  print x
7  print y
8
9  print "I said: %r." % x
10 print "I also said: '%s'." % y
11
12 hilarious = False
13 joke_evaluation = "Isn't that joke so funny?! %r"
14
15 print joke_evaluation % hilarious
16
17 w = "This is the left side of..."
18 e = "a string with a right side."
19
20 print w + e
```

Exercise: Parameters, Unpacking, Variables


```
1  from sys import argv
2
3  script, first, second, third = argv
4
5  print "The script is called:", script
6  print "Your first variable is:", first
7  print "Your second variable is:", second
8  print "Your third variable is:", third
```

Exercise: Read Files


```
1  from sys import argv
2
3  script, filename = argv
4
5  txt = open(filename)
6
7  print "Here's your file %r:" % filename
8  print txt.read()
9
10 print "Type the filename again:"
11 file_again = raw_input("> ")
12
13 txt_again = open(file_again)
14
15 print txt_again.read()
```

A few fancy things are going on in this file, so let's break it down real quick:

Lines 1-3 uses `argv` to get a filename. Next we have line 5 where we use a new command `open`. Right now, run `pydoc open` and read the instructions. Notice how like your own scripts and `raw_input`, it takes a parameter and returns a value you can set to your own variable. You just opened a file.

Line 7 prints a little message, but on line 8 we have something very new and exciting. We call a function on `txt` named `read`. What you get back from `open` is a `file`, and it also has commands you can give it. You give a file a command by using the `.` (dot or period), the name of the command, and parameters. Just like with `open` and `raw_input`. The difference is that `txt.read()` says, "Hey `txt`! Do your read command with no parameters!"

Exercise: Read and Write files


```
1  from sys import argv
2
3  script, filename = argv
4
5  print "We're going to erase %r." % filename
6  print "If you don't want that, hit CTRL-C (^C)."
7  print "If you do want that, hit RETURN."
8
9  raw_input("?")
10
11 print "Opening the file..."
12 target = open(filename, 'w')
13
14 print "Truncating the file.  Goodbye!"
15 target.truncate()
16
17 print "Now I'm going to ask you for three lines."
18
19 line1 = raw_input("line 1: ")
20 line2 = raw_input("line 2: ")
21 line3 = raw_input("line 3: ")
22
23 print "I'm going to write these to the file."
24
25 target.write(line1)
26 target.write("\n")
27 target.write(line2)
28 target.write("\n")
29 target.write(line3)
30 target.write("\n")
31
32 print "And finally, we close it."
33 target.close()
```